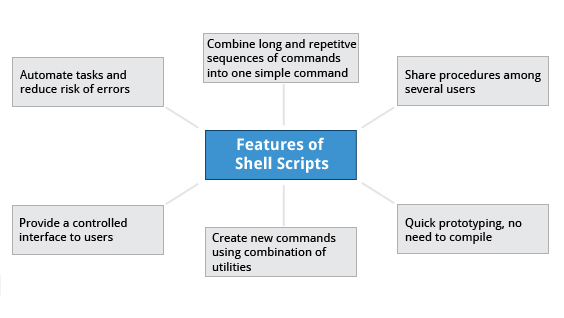
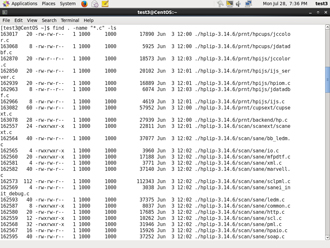
**Introduction to Scripts**



Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence. If you only need to do it once, you can just type a sequence of commands at a terminal. However, if you need to do this multiple times, automation is the way to go. In order to automate sets of commands you’ll need to learn how to write **shell scripts**, the most common of which are used with **bash**. The graphic illustrates several of the benefits of deploying scripts.

**Introduction to Shell Scripts**



Remember from our earlier discussion, a **shell** is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in. For example typing: find . -name "\*.c" -ls at the command line accomplishes the same thing as executing a script file containing the lines:

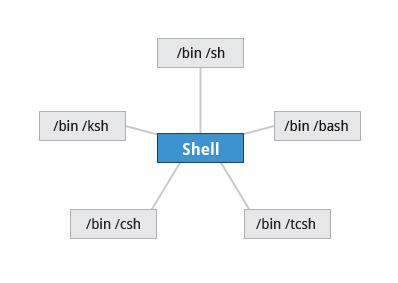
#!/bin/bash

find . -name "\*.c" -ls

The #!/bin/bash in the first line should be recognized by anyone who has developed any kind of script in UNIX environments. The first line of the script, that starts with #!, contains the full path of the command interpreter (in this case /bin/bash) that is to be used on the file. As we will see on the next screen, you have a few choices depending upon which scripting language you use.

Click the image to view an enlarged version.

**Command Shell Choices**



The command **interpreter** is tasked with executing statements that follow it in the script. Commonly used interpreters include: /usr/bin/perl, /bin/bash, /bin/csh, /usr/bin/python and /bin/sh.

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command-line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

/bin/sh

/bin/bash

/bin/tcsh

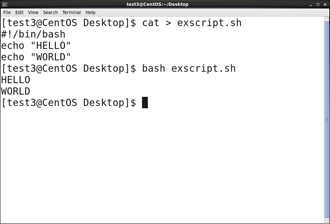
/bin/csh

/bin/ksh

Most Linux users use the default **bash** shell, but those with long UNIX backgrounds with other shells may want to override the default.

Click the link to download [UNIX Shell.pdf](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/Chap14_UNIXShell.pdf) to read more about the UNIX Shell.

**bash Scripts**

Let's write a simple **bash** script that displays a two-line message on the screen. Either type

$ cat > exscript.sh

#!/bin/bash

echo "HELLO"

echo "WORLD"

and press **ENTER** and **CTRL-D** to save the file, or just create exscript.sh in your favorite text editor. Then, type chmod +x exscript.sh to make the file executable. (The chmod +x command makes the file executable for all users.) You can then run it by simply typing ./exscript.sh or by doing:

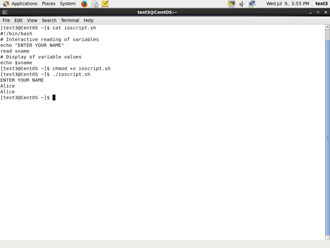
$ bash exscript.sh

HELLO

WORLD

Note if you use the second form, you don't have to make the file executable.

**Interactive Example Using bash Scripts**

Now, let's see how to create a more interactive example using a **bash** script. The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, sname. We can reference the value of a shell variable by using a $ in front of the variable name, such as $sname. To create this script, you need to create a file named ioscript.sh in your favorite editor with the following content:

#!/bin/bash

# Interactive reading of variables

echo "ENTER YOUR NAME"

read sname

# Display of variable values

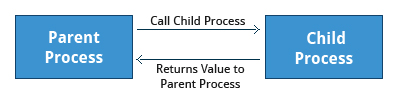
echo $sname

Once again, make it executable by doing chmod +x ioscript.sh.

In the above example, when the script ./ioscript.sh is executed, the user will receive a prompt ENTER YOUR NAME. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

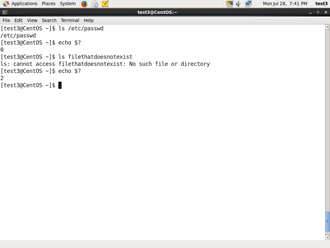
Additional note: The hash-tag/pound-sign/number-sign (#) is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment).

**Return Values**



All shell scripts generate a **return value** upon finishing execution; the value can be set with the exit statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure.

**Viewing Return Values**



As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute **ls** on a file that exists and one that doesn't, as shown in the following example, where the return value is stored in the environment variable represented by $?:

$ ls /etc/passwd

/etc/ passwd

$ echo $?

0

In this example, the system is able to locate the file /etc/passwd and returns a value of 0 to indicate success; the return value is always stored in the $? environment variable. Applications often translate these return values into meaningful messages easily understood by the user.

**Basic Syntax and Special Characters**

Scripts require you to follow a standard language **syntax**. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within **bash** scripts:

|  |  |
| --- | --- |
| **Character** | **Description** |
| # | Used to add a comment, **except** when used as \#, or as #! when starting a script |
| \ | Used at the end of a line to indicate continuation on to the next line |
| ; | Used to interpret what follows as a new command |
| $ | Indicates what follows is a variable |

Note that when # is inserted at the beginning of a line of commentary, the whole line is ignored.

# This line will not get executed.

**Splitting Long Commands Over Multiple Lines**



Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called **chaining of commands**.

The **concatenation operator** (**\**) is used to concatenate large commands over several lines in the shell.

For example, you want to copy the file **/var/ftp/pub/userdata/custdata/read** from **server1.linux.com** to the **/opt/oradba/master/abc** directory on **server3.linux.co.in**. To perform this action, you can write the command using the \ operator as:

scp abc@server1.linux.com:\

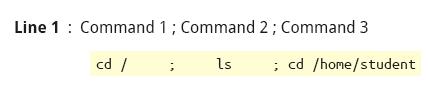
/var/ftp/pub/userdata/custdata/read \

abc@server3.linux.co.in:\

/opt/oradba/master/abc/

The command is divided into multiple lines to make it look readable and easier to understand. The \ operator at the end of each line combines the commands from multiple lines and executes it as one single command.

**Putting Multiple Commands on a Single Line**



Sometimes you may want to group multiple commands on a single line. The ; (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

The three commands in the following example will all execute even if the ones preceding them fail:

$ make ; make install ; make clean

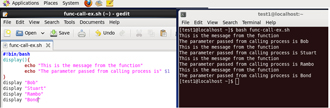
However, you may want to abort subsequent commands if one fails. You can do this using the && (and) operator as in:

$ make && make install && make clean

If the first command fails the second one will never be executed. A final refinement is to use the || (or) operator as in:

$ cat file1 || cat file2 || cat file3

**Functions**



A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines.** Using functions in scripts requires two steps:

1. Declaring a function

2. Calling a function

The function declaration requires a name which is used to invoke it. The proper syntax is:

function\_name () {

command...

}

For example, the following function is named display:

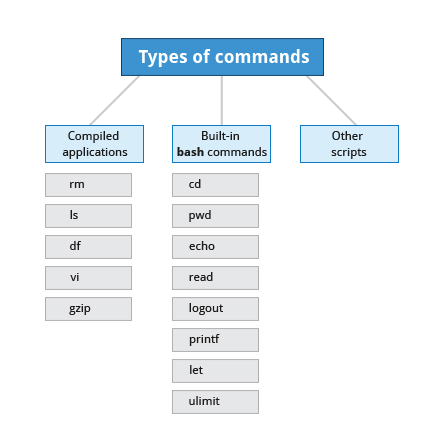
display () {

echo "This is a sample function"

}

The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function. The first argument can be referred to as $1, the second as $2, etc.

**Built-in Shell Commands**



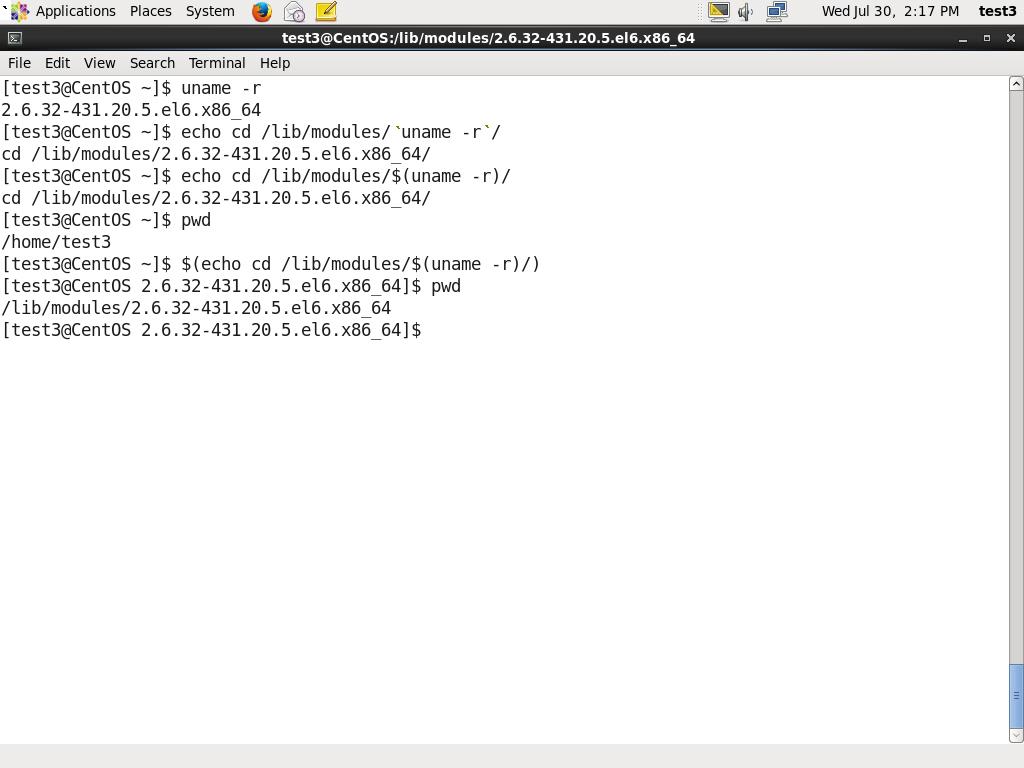
Shell scripts are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:

* Compiled applications
* Built-in **bash** commands
* Other scripts

Compiled applications are binary executable files that you can find on the filesystem. The shell script always has access to compiled applications such as **rm**, **ls**, **df**, **vi**, and **gzip**.

**bash** has many **built-in** commands which can only be used to display the output within a terminal shell or shell script. Sometimes these commands have the same name as executable programs on the system, such as **echo** which can lead to subtle problems. **bash** built-in commands include and cd, pwd, echo, read, logout, printf, let, and ulimit.

A complete list of **bash** built-in commands can be found in the **bash man** page, or by simply typing help.

**Command Substitution**

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

* By enclosing the inner command with backticks (`)
* By enclosing the inner command in $( )

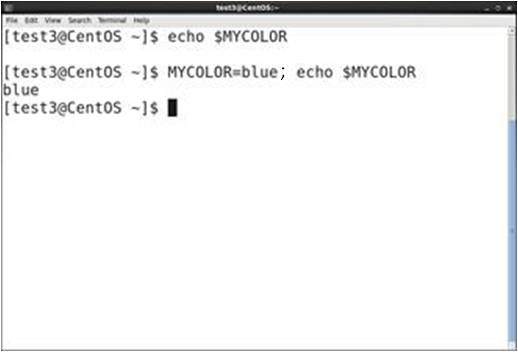
No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.

Virtually any command can be executed this way. Both of these methods enable command substitution; however, the $( ) method allows command nesting. New scripts should always use this more modern method. For example:

$ cd /lib/modules/$(uname -r)/

In the above example, the output of the command "uname –r" becomes the argument for the cd command.

**Environment Variables**

Almost all scripts use **variables** containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Many applications use such **environment variables** (covered in the "User Environment" chapter) for supplying inputs, validation, and controlling behaviour.

Some examples of standard environment variables are HOME, PATH, and HOST. When referenced, environment variables must be prefixed with the $ symbol as in $HOME. You can view and set the value of environment variables. For example, the following command displays the value stored in the PATH variable:

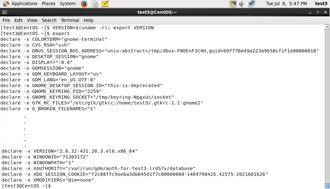
$ echo $PATH

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the MYCOLOR variable to blue:

$ MYCOLOR=blue

You can get a list of environment variables with the **env**, **set**, or **printenv** commands.

**Exporting Variables**



By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the **export** statement as in:

export VAR=value

or

VAR=value ; export VAR

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

**Script Parameters**

Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers as in:

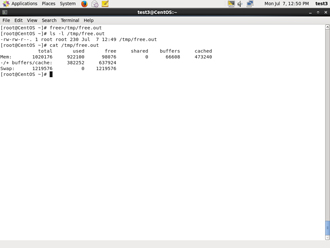
$ ./script.sh /tmp

$ ./script.sh 100 200

Within a script, the parameter or an argument is represented with a $ and a number. The table lists some of these parameters.

|  |  |
| --- | --- |
| **Parameter** | **Meaning** |
| $0 | Script name |
| $1 | First parameter |
| $2, $3, etc. | Second, third parameter, etc. |
| $\* | All parameters |
| $# | Number of arguments |

**Output Redirection**

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output **redirection**.

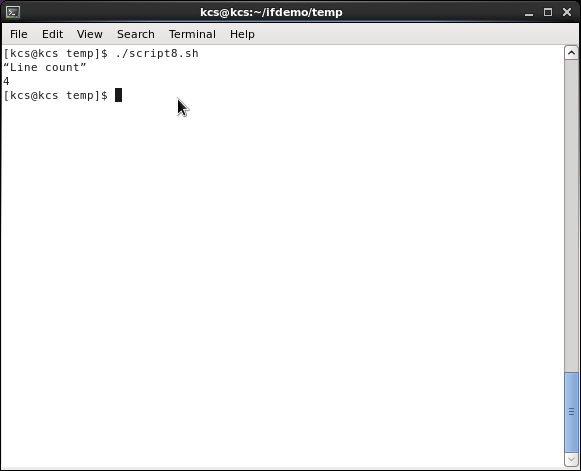
The > character is used to write output to a file. For example, the following command sends the output of **free** to the file /tmp/free.out:

$ free > /tmp/free.out

To check the contents of the /tmp/free.out file, at the command prompt type cat /tmp/free.out.

Two > characters (>>) will append output to a file if it exists, and act just like > if the file does not already exist.

**Input Redirection**



Just as the output can be redirected **to** a file, the input of a command can be read **from** a file. The process of reading input from a file is called input redirection and uses the < character. If you create a file called script8.sh with the following contents:

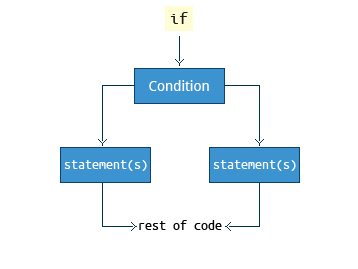
#!/bin/bash

echo “Line count”

wc -l < /temp/free.out

and then execute it with chmod +x script8.sh ; ./script8.sh, it will count the number of lines from the **/temp/free.out** file and display the results.

**The if Statement**



Conditional decision making using an if statement, is a basic construct that any useful programming or scripting language must have.

When an if statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

* Numerical or string comparisons
* Return value of a command (0 for success)
* File existence or permissions

In compact form, the syntax of an if statement is:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

A more general definition is:

if condition

then

statements

else

statements

fi

The following if statement checks for the /etc/passwd file, and if the file is found it displays the message

/etc/passwd exists.:

if [ -f /etc/passwd ]

then

echo "/etc/passwd exists."

fi

Notice the use of the square brackets ([]) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly; we will discuss these other tests.

In modern scripts you may see doubled brackets as in[[ -f /etc/passwd ]]. This is not an error. It is never wrong to do so and it avoids some subtle problems such as referring to an empty environment variable without surrounding it in double quotes; we won't talk about this here.

**Testing for Files**

You can use the if statement to test for file attributes such as:

* File or directory existence
* Read or write permission
* Executable permission

For example, in the following example:

if [ -f /etc/passwd ] ; then

ACTION

fi

the if statement checks if the file /etc/passwd is a regular file.

Note the very common practice of putting **“; then”** on the same line as the **if** statement.

**bash** provides a set of **file conditionals**, that can used with the if statement, including:

|  |  |
| --- | --- |
| **Condition** | **Meaning** |
| -e file | Check if the file exists. |
| -d file | Check if the file is a directory. |
| -f file | Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.) |
| -s file | Check if the file is of non-zero size. |
| -g file | Check if the file has sgid set. |
| -u file | Check if the file has suid set. |
| -r file | Check if the file is readable. |
| -w file | Check if the file is writable. |
| -x file | Check if the file is executable. |

**Example of Testing of Strings**

You can use the if statement to compare strings using the operator == (two equal signs). The syntax is as follows:

if [ string1 == string2 ] ; then

ACTION

fi

Let’s now consider an example of testing strings.

In the example illustrated here, the if statement is used to compare the input provided by the user and accordingly display the result.

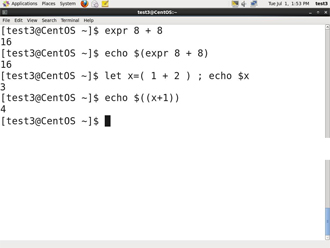
**Numerical Tests**

You can use specially defined operators with the if statement to compare numbers. The various operators that are available are listed in the table.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| -eq | Equal to |
| -ne | Not equal to |
| -gt | Greater than |
| -lt | Less than |
| -ge | Greater than or equal to |
| -le | Less than or equal to |

The syntax for comparing numbers is as follows:

**Arithmetic Expressions**

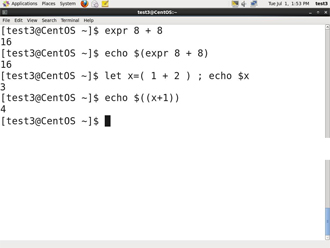


Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

* Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows:
* expr 8 + 8
* echo $(expr 8 + 8)
* Using the $((...)) syntax: This is the built-in shell format. The syntax is as follows:
* echo $((x+1))
* Using the built-in shell command let. The syntax is as follows:
* let x=( 1 + 2 ); echo $x

In modern shell scripts the use of **expr** is better replaced with var=$((...))

**Arithmetic Expressions**



Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

* Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows:
* expr 8 + 8
* echo $(expr 8 + 8)
* Using the $((...)) syntax: This is the built-in shell format. The syntax is as follows:
* echo $((x+1))
* Using the built-in shell command let. The syntax is as follows:
* let x=( 1 + 2 ); echo $x

In modern shell scripts the use of **expr** is better replaced with var=$((...))